6.1420/6.S974 Fixed-Parameter and Fine-Grained Complexity **MIT**
Lectures 3: Some Improved SAT Algorithms **September 15, 2024**
Virginia Vassilevska Williams and Ryan Williams (notes by Ryan)

Today: Previously, we discussed ETH and SETH. Both of these are conjectures about the time complexity of SAT. But to really understand why people believe in these conjectures, it's important to know what techniques *are* known for solving SAT faster. So for the next two lectures, we will discuss the best known theoretical algorithms and techniques for solving SAT in the worst case. (In practice, SAT is often solved very efficiently. But there are instances in practice that make SAT solvers trip up, and run in exponential time.)

Think about how to do the exercises below! They'll help you understand the material. (But you don't have to turn in solutions to them.)

1 Randomized reduction: a first start

We'll start with a simple algorithm that beats 2^n time in a special case of 3SAT. It will illustrate some useful principles in some of the algorithms to come. In the following, a 3-clause is a clause with three literals.

Theorem 1.1 3-SAT *can be solved in* $O^\star((3/2)^t)$ *randomized time on formulas with at most* t *3-clauses. In particular, our randomized algorithm always outputs "UNSAT" on unsatisfiable formulas, and outputs "SAT" on satisfiable formulas with probability greater than* $1 - 10^{-9}$.

Note, some 3-SAT instances could have many 2-clauses and a small number of (say, $O(\log n)$) 3-clauses. The above theorem shows that these instances could be solved quickly with randomness. The algorithm is basically a *randomized reduction* from 3SAT to 2SAT.

Proof. First we'll give an algorithm **RANDO** which takes a formula F with at most t 3-clauses. Then we'll explain why **RANDO** works.

```
RANDO($F$):
Repeat for $20 \cdot (3/2)^t$ trials:
    Let $F'$ be the set of 2-CNF and 1-CNF clauses in $F$.
    For every 3-clause $C = (\ell_1 \vee \ell_2 \vee \ell_3)$ in $F$,
        Randomly choose an $\ell_i$ and remove $\ell_i$ from $C$, forming a new 2-clause $C'$. Add $C'$ to $F'$.
    End for
    Solve the resulting 2-SAT instance $F'$ in polynomial time. If $F'$ is satisfiable then return "SAT".
End repeat
Return "UNSAT".
```

Exercise 1.1 *Hmm... how do you solve* 2-SAT *in polynomial time? (There are multiple ways you could prove this.)*

Now for the analysis of **RANDO**. The key observation is this: if any F' is satisfiable by some assignment A, then F is also satisfiable by A, because F' is a *restriction* to F (its clauses are only shorter than those of F).

First, if F is unsatisfiable, then every F' computed by **RANDO** is also unsatisfiable, and the algorithm will never return "SAT".

Second, suppose F is satisfiable, and let A be a satisfying assignment to it.

Claim 1.1 *For all clauses C, $Pr[A \text{ satisfies } C'] \geq 2/3$.*

Proof of Claim: In the worst case, exactly one literal ℓ in C is satisfied by A. (In fact, if more than one literal in C is satisfied by A, then A satisfies C' with probability 1![1]) This literal ℓ is removed with probability $1/3$. And if you remove any other literal instead, the remaining clause is still satisfied.
QED

Since each literal removed is an independent choice, we have:

$$Pr[F' \text{ is SAT by } A] \geq (2/3)^t.$$

In **RANDO**, we repeat the inner loop for $r = 20 \cdot (3/2)^t$ trials. Therefore

$$Pr[\text{No } F' \text{ is SAT by } A \text{ over all trials}] \leq (1-(2/3)^t)^r = (1-(2/3)^t)^{20(3/2)^t} \leq \exp(-20 \cdot (2/3)^t \cdot (3/2)^t) = \exp(-20).$$

(Here, we applied the useful inequality $(1-x) \leq \exp(-x)$.) Therefore, when the algorithm reports "UNSAT", the probability it is wrong is less than $e^{-20} < 10^{-9}$. □

Note: as far as I know, the best known algorithm of this kind (in terms of the number of 3-clauses t) is deterministic and runs in $O^\star(1.3645^t)$ time [BE05].

2 Algorithms which beat 2^n time in general

Now we turn to algorithms which really do beat 2^n time for k-SAT. First, we give some simple improvements over exhaustive search. We'll start with branching (a.k.a. backtracking) algorithms, as they are very natural. In a branching algorithm, the idea is that you try to cleverly pick variables to assign to true or false. As you assign them, all of their occurrences in the formula get removed, and their removal simplifies the formula. (If you are clever about how you pick them, you could simplify the formula considerably.) Then you recurse on the simplified formula.

Among the algorithms we'll see, the branching paradigm is most like actual real-life SAT solvers. (In the 1990s, local search was the fastest, but in the early 2000s people began engineering fast branching algorithms coupled with additional heuristics, and they've been the fastest ever since, with many technical refinements and improvements added over the last 20 years.)

Theorem 2.1 *k-SAT on n variables is in $2^{n-n/O(k2^k)} \cdot poly(n)$ time.*[2]

Proof. As promised, this will be a backtracking/branching algorithm. The key idea is this: whenever we have a clause of ℓ literals, there are only $2^\ell - 1$ satisfying assignments to the clause, even though there are 2^ℓ total assignments to the clause. This difference of one assignment is enough to get some running time improvement!

```
Algorithm A($F$):    // $F$ is a $k$-CNF formula
If $F$ has no clauses, return SAT.
If $F$ contains an empty clause (a clause with no literals), return UNSAT.
Take the shortest clause in $F$, call it $C = (x_1 \vee \ldots \vee x_\ell)$.
For all $2^\ell - 1$ satisfying assignments $a \in \{0,1\}^\ell$ to $C$,
```

[1] Note you could have interpreted ! as factorial, and the statement is still true. You could have even interpreted the footnote symbol as an exponent of 1, and the statement is still true.

[2] That is, there are fixed constants $c, d > 0$ such that for all k, k-SAT on n variables is solvable in $2^{n-n/ck2^k} \cdot n^d$ time.

```
    Call A($F|_{x_1=a_1,\ldots,x_\ell=a_\ell}$).
      // this notation just means $x_1,\ldots,x_\ell$ are replaced by the bits in $a$, so $F|_{x_1=a_1,\ldots,x_\ell=a_\ell}$ has $\ell$ fewer variables;
      // also, any already satisfied clauses are removed.
If one of these calls returns SAT, then return SAT. Otherwise return UNSAT.
```

Analysis: In the worst case, the shortest clause C is always of length k. (It couldn't be any longer.) In that case, our running time recurrence is $T(n) \leq (2^k - 1)T(n-k) + O(\text{poly}(n))$. This easily solves to $T(n) \leq (2^k - 1)^{n/k}\text{poly}(n)$. Using the inequality $1 - x \leq e^{-x}$, we find that

$$(2^k - 1)^{n/k} = 2^n(1 - 1/2^k)^{n/k} \leq 2^n e^{-n/(k2^k)},$$

and we are done. □

Exercise 2.1 *Convince yourself that we can already conclude the following:*
For all k, there is a $\delta < 1$ such that k-SAT can be solved in $O(2^{\delta n})$ time.
(Wait, what happened to the poly(n) factor in the running time? Why can we omit it?)
Note the difference between the above statement, and SETH.

We can improve the dependence on k slightly with a cleverer branching:

Theorem 2.2 *k-SAT on n variables is in $2^{n-n/O(2^k)}$ time.*

Proof. This is based on the branching/backtracking algorithm of Monien and Speckenmeyer [MS85].

```
Algorithm A($F$): // $F$ is a $k$-CNF formula
If $F$ has no clauses, return SAT. If $F$ has an "empty" clause (clause set false), return UNSAT.
If $F$ is 2-CNF, solve SAT in polytime and return the answer.
If there is a 1-CNF clause $(x)$, call A on $F|_{x=1}$.
Take shortest clause $(x_1 \vee \ldots \vee x_L)$.
Call A on $F|_{x_1=1}, F|_{x_1=0,x_2=1}, \ldots, F|_{x_1=0,\ldots,x_{i-1}=0,x_i=1}, \ldots, F|_{x_1=0,x_2=0,\ldots,x_L=1}$.
If any of the $L$ calls says SAT, then return SAT; else return UNSAT.
```

Recurrence for the running time: $T(n) \leq \sum_{i=1}^{k} T(n-i) + O(\text{poly}(n))$.

Exercise 2.2 *Why is this the running time recurrence? Why is this algorithm correct?*

How to solve it? We will guess that $T(n) = 2^{\alpha n}$ for some parameter $\alpha > 0$, and we'll try to find $\alpha < 1$ that satisfies the recurrence. Inductively, we want the following to be true:

$$T(n) \leq \sum_{i=1}^{k} T(n-i) \leq \sum_{i=1}^{k} 2^{\alpha(n-i)} \leq 2^{\alpha n}.$$

In particular, we want $\sum_{i=1}^{k} 2^{\alpha(n-i)} \leq 2^{\alpha n}$. Dividing both sides by $2^{\alpha n}$, this is $\sum_{i=1}^{k} 2^{-\alpha i} \leq 1$. We need to find an α that will satisfy this inequality. By the usual expression for the sum of a geometric series, this inequality is equivalent to

$$(1 - 2^{-\alpha(k+1)})/(1 - 2^{-\alpha}) - 1 \leq 1.$$

Manipulating this around, we get that the above is equivalent to

$$(1-2^{-\alpha(k+1)})/(1-2^{-\alpha}) \leq 2 \iff 1-2^{-\alpha(k+1)} \leq 2-2^{1-\alpha} \iff 2^{1-\alpha}-2^{-\alpha(k+1)} \leq 1.$$

Being incredible guessers, let's try $\alpha = 1 - \log_2(e)/(5 \cdot 2^k)$. Then $2^{1-\alpha} - 2^{-\alpha(k+1)} = 2^{\log(e)/(10 \cdot 2^k)} - 2^{-(k+1)(1-\log(e)/(5 \cdot 2^k))}$ which equals

$$e^{1/(5\cdot 2^k)} - 2^{-(k+1)(1-\log(e)/(5\cdot 2^k))} < 1 + 3/(10 \cdot 2^k) - 2^{-(k+1)(1-\log(e)/(5\cdot 2^k))} < 1,$$

where we used the inequalities $e^x < 1 + 3x/2$ for all $x \in (0,1)$ and $2^{(k+1)\log(e)/(5\cdot 2^k)}/2^{k+1} > 3/(10 \cdot 2^k)$ for all $k \geq 1$. □

Exercise 2.3 *Is there a cleaner derivation of $\alpha \leq 1 - O(1/2^k)$?* **:)**

In general, the following is useful for analyzing backtracking algorithms:

Theorem 2.3 *Every recurrence of the form*

$$T(n) \leq T(n-k_1) + T(n-k_2) + \cdots + T(n-k_i) + O(\mathit{poly}(n))$$

has as a solution

$$T(n) \leq O(r(k_1,\ldots,k_i)^n \cdot \mathit{poly}(n)),$$

where $r(k_1,\ldots,k_i)$ is a positive root of the expression $P(x) = 1 - \sum_{j=1}^{i} x^{-k_j}$.

For example, consider

$$T(n) \leq T(n-1) + T(n-2) + O(\text{poly}(n)).$$

By the above theorem, we want to find positive x satisfying $1 - 1/x - 1/x^2 = 0$. (Ideally, we want the smallest such x too, to optimize the running time bound!) This is equivalent to $x^2 - x - 1 = 0$ which is the same as $x(x-1) = 1$. Solutions for x are $x = 1.618\ldots, -.618\ldots$, so $T(n) \leq O(1.618^n)$.

Exercise 2.4 *Think about how you might prove the above theorem.*

3 Improved Algorithms for k-SAT

In this section, we will give the shortest proof we know that k-SAT has an $2^{n-n/O(k)}$-time algorithm.

Theorem 3.1 *k-SAT on n variables is in $2^{n-n/O(k)}$ time.*

This is the essentially the best dependence on k in the exponent that we know (to date). Improving on the exponent of $n(1-1/O(k))$ is a major open problem! For example, is there an algorithm with exponent $n(1-\log(k)/O(k))$? In the literature, there is a hypothesis called Super-SETH [VW19] which posits that there is no unbounded function $f : \mathbb{N} \to \mathbb{N}$ such that k-SAT can be solved in $2^{n-f(k)n/k}$ time. (Of course, Super-SETH implies SETH.) Ryan probably gives $\leq 10\%$ likelihood that Super-SETH is true. (For other estimates, see [Wil19].)

Local search and random walks. Schoening [Sch99, Sch02] obtained an improved solution of k-SAT using an entirely different strategy. It is based on an earlier local search / random walk algorithm for solving 2-SAT, due to Papadimitriou [Pap91].

Theorem 3.2 ([Pap91]) *There is a randomized algorithm for* 2-SAT *running in poly*(n) *time.*

Here we sketch the proof. The algorithm is as follows:

```
LS($F$):
Let $A$ be a random assignment to the $n$ variables of $F$.
Repeat for $100n^2$ times:
    If $F$ is SAT by $A$, return "SAT".
    Else pick a clause $c$ that $A$ falsifies
        and pick a variable $v$ in $c$ at random.
        Flip the value of $v$ in $A$.
End repeat
Return "UNSAT".
```

Let us prove that **LS** works with good probability. Clearly if **LS** returns "SAT", then F is SAT (it finds a satisfying assignment).

Claim 3.1 *Suppose F is SAT. Pr[***LS***(F) returns "UNSAT"] $< 1/3$.*

We'll sketch the proof of this claim. We will associate the behavior of the local search algorithm with a *random walk on a line graph*. Let A^* be a satisfying assignment to F. (In the worst case for us, there is only one satisfying assignment.) Consider a line graph on $n+1$ nodes labeled by $\{0, \ldots, n\}$.

Exercise 3.1 *Imagine a picture of a line graph on $n+1$ nodes, drawn here*

Each node on the line is associated with a subset of the n-variable Boolean assignments. In particular, node $i \in \{0, 1, \ldots, n\}$ corresponds to the set of assignments $A' \in \{0,1\}^n$ such that $h(A^*, A') = i$, where

$$h(x, y) = \text{ Hamming distance between } x \text{ and } y = \text{number of bits in which } x \text{ and } y \text{ differ.}$$

Therefore:

- Node 0 corresponds to A^*,
- Node 1 corresponds to the n assignments that are like A^*, except one of their bits are flipped, and in general
- Node i corresponds to $\binom{n}{i}$ assignments that have i bits different from A^*.

How will this graph correspond to the algorithm **LS**? At each step, the algorithm **LS** is holding an assignment A in memory, associated with some node i on the line. If A corresponds to node 0, then $A = A^*$, and **LS** halts and says "SAT". When we are not at node 0, we would like for **LS** to have a good chance of "moving towards" node 0 by flipping bits in its current assignment A.

Remember **LS** picks a clause that isn't satisfied, randomly picks one of the two variables, and flips its value. Now since it's a 2-CNF, each clause has two variables, and at least one of them is satisfied by A^*. So there is a probability at least $1/2$ of moving one bit closer to A^* when we flip a bit, and probability at most $1/2$ of moving one bit *away*.

Therefore we can think of the LS algorithm as performing a *random walk* on this line graph: it takes one step towards node 0 with probability at least $1/2$, and one step away with probability at most $1/2$. Now we want to know how long it will take for **LS** to reach node 0. It turns out that, with high probability, **LS** will reach node 0 after $100n^2$ steps. Here we'll only give intuition for the proof; we'll go in more detail for the case of 3-SAT.

Proof Intuition: There are multiple ways you can try to prove this. One way is to use **lower bounds** on the tail of the binomial distribution. (Note: Almost all results you learn in school about tails of binomials are upper bounds, not lower bounds.) Suppose we start at any node of the line graph. View the random walk as flipping a coin that comes up heads or tails: if it's heads we decrease the node number, and if it's tails we increase the node number (if the node number is already n, it stays the same). The coin comes up heads with probability at least $1/2$. We want to know how many times we need to flip the coin, before we can expect the number of heads to exceed the number of tails by n. (After that point, we definitely will have reached node 0 during the walk.)

For $i = 1, 2, \ldots$, define a random variable X_i which is 0 with probability $1 - p$ and 1 with probability p, where $p \geq 1/2$, and define $X_M = \sum_{i=1}^{M} X_i$. Standard probability results tell us that $E[X_M] = pM$ and Stddev$(X_M) = \sqrt{p(1-p)M}$.

Claim 3.2 *For every $\varepsilon \in (0, 1)$ there is a $C_\varepsilon \geq 1$ such that*

$$\Pr[\textit{for some } M = 1, \ldots, C_\varepsilon n^2, X_M \geq n + E[X_M]] \geq 1 - \varepsilon.$$

(Note, this claim is lower bounding $X_M - E[X_M]$.) This claim implies that the random walk in **LS**, when repeated for C_ε times, will produce a satisfying assignment with probability at least $1-\varepsilon$. (When $E[X_M] = X_M$ and $p = 1/2$, the number of heads equals the number of tails, so in the claim we are looking at the case where the number of heads exceeds the number of tails by at least n.) The idea is that for $T = Cn^2$ where C is sufficiently large, the standard deviation of X_T is greater than n, and the probability that X_M exceeds $n + E[X_M]$ for some $M = 1, \ldots, T$ becomes close to 1 (using properties of binomials). [3]

Exercise 3.2 *If you're bored or fascinated at this point, you could try your hand at proving the claim. Or you could try to analyze the algorithm in another way. But this is optional. Tim Roughgarden walks through a proof that the random walk process works in $O(n^2)$ steps, here:* https://www.coursera.org/lecture/algorithms-npcomplete/random-walks-on-a-line-kRmJe

The next natural question is: **Why doesn't this algorithm work for 3-SAT?**

Well, it will "work", but the random walk probabilities get screwed up! For 3-SAT, we will only have probability $1/3$ of flipping the correct variable and moving towards node 0, and probability $2/3$ of moving away, so we are *much* more likely to drift away from 0 than to move towards 0. That's not good at all. Nevertheless, Schoening found an adaptation of the algorithm which leads to a less-than-2^n running time for 3-SAT.

3.1 Schoening's algorithm and its generalization

We will begin with a simplified version of the random walk algorithm that works for k-SAT. As you can see in the below theorem, it does give the best known asymptotic dependence on k in the exponent.

Theorem 3.3 *k-SAT can be solved in $2^{n-n/O(k)}$ time.*

[3] Note: You can also analyze the random walk by choosing a different random variable, which is more standard. For $i = 1, 2, \ldots$, define a random var X_i which is -1 with probability $1 - p$ and 1 with probability p, where $p \geq 1/2$, and define $X_M = \sum_{i=1}^{M} X_i$. Standard probability results tell us that $E[X_M] = (2p-1)M$ and Var$(X_M) = \sum_{j=1}^{M} E[X_i]^2 + \sum_{i \neq j} E[X_i \cdot X_j] = M + (M^2 - M)(p^2 + (1-p)^2 - 2p(1-p)) = M^2 + (M^2 - M)(4p^2 - 4p)$, so that Stddev$(X_M) = \sqrt{M^2 - 4p(1-p)(M^2 - M)} \in [\sqrt{M}, M]$. Then, you want to analyze the probability that X_M exceeds n, instead of $n + E[X_M]$.

Proof. We will first present the inner loop of the algorithm, which succeeds with some probability at finding a satisfying assignment. Later, we'll repeat it for some number of times. In general, suppose we have a procedure that always says "UNSAT" when F is unsatisfiable, and says "SAT" when F is satisfiable with probability at least P. Then, if we repeat this procedure for $10/P$ trials, and say "UNSAT" only if none of the trials returned "SAT", this new algorithm will be correct with probability $\geq 1 - \exp(-10)$, as we showed earlier (in our first algorithm). Thus, we will focus on giving procedures that are fast and are correct with exponentially low probability, but when we run them an exponential number of times, they become correct with constant probability. Here is such an algorithm for k-SAT:

```
Schon($F$):
Choose random assignment $A$.
Repeat for $n/k$ times:
    If $A$ satisfies $F$ return "SAT"
    If $A$ does not satisfy $F$, then
        Let $C$ be a falsified clause of $F$.
        Pick a random variable in $C$. Flip its value in $A$.
End repeat.
Return "UNSAT".
```

Analysis of algorithm. Let A^* be a SAT assignment to F, as before. We will show **Schon** outputs a satisfying assignment (when one exists) with probability at least

$$\frac{1}{2^n \cdot e^{-n/k+n/k^2} \cdot (n+1)}.$$

Claim 3.3 *There is a $c > 0$ such that $Pr[A^*$ found by* **Schon**$(F)] \geq 1/(2^n \cdot e^{-n/k+n/k^2} \cdot (n+1)) \geq 1/2^{n-n/ck}$.

So, if we repeat **Schon** for $t = 10 \cdot 2^{n-n/ck}$ times, we'll have a high probability of finding a SAT assignment using it.

Proof of Claim: Define event E to be: *Randomly chosen assignment A is within n/k Hamming distance of A^*.* Then

$$\Pr[E] \geq \binom{n}{n/k} / 2^n.$$

This is because there are $\binom{n}{n/k}$ different strings A' such that $h(A^*, A') = n/k$. We also have

$$\Pr[\text{Local search for } n/k \text{ steps finds } A^* \mid E] \geq \left(\frac{1}{k}\right)^{n/k},$$

because for each variable that is randomly chosen in **Schon**, there's at least a $1/k$ chance that we chose a variable that is flipped to its correct value. And if we started at an assignment that is n/k bits away from A^*, and we flip all n/k variables correctly in our local search, we will be at assignment A^*. Therefore

$$\begin{aligned}\Pr[A^* \text{ is found by } \textbf{Schon}] &\geq \Pr[E] \cdot \Pr[\text{Local search for } n/k \text{ steps finds } A^* \mid E] \\ &\geq \frac{\binom{n}{n/k}}{2^n} \cdot \left(\frac{1}{k}\right)^{n/k}.\end{aligned}$$

We now claim that

$$\frac{\binom{n}{n/k}}{2^n} \cdot \left(\frac{1}{k}\right)^{n/k} \geq \frac{1}{2^n \cdot e^{-n/k+n/k^2} \cdot (n+1)}.$$

Once we prove that, we'll have completed the proof of the theorem.

(Note: The rest of this proof is really optional material. But it doesn't take too long to get through it, and you might learn some new extremal combinatorics while you're at it.)

The claimed inequality follows from applying other nice inequalities in the right way, namely:

1. $1 - x \leq e^{-x}$,
2. $\binom{n}{\alpha n} \geq 2^{H(\alpha)n}/(n+1)$, where $H(a) := a \log_2(1/a) + (1-a)\log_2(1/(1-a))$ is the binary entropy function.

By inequality 2, noting that $2^{H(1/k)n} = k^{n/k} \cdot (1/(1-1/k))^{n-n/k}$, we have

$$\frac{\binom{n}{n/k}}{2^n} \cdot \left(\frac{1}{k}\right)^{n/k} \geq \frac{k^{n/k} \cdot (1/(1-1/k))^{n-n/k}}{k^{n/k} 2^n (n+1)}.$$

Cancelling out $k^{n/k}$ factors, we get

$$\frac{k^{n/k} \cdot (1/(1-1/k))^{n-n/k}}{k^{n/k} 2^n (n+1)} = \frac{1}{2^n \cdot (1-1/k)^{n-n/k}(n+1)}.$$

Finally, applying inequality 1, we obtain

$$\frac{1}{2^n \cdot (1-1/k)^{n-n/k}(n+1)} \geq \frac{1}{2^n \cdot e^{-n/k+n/k^2} \cdot (n+1)}.$$

□

What is the key to the speedup in the **Schon** algorithm? Here's what's happening: instead of taking $O^*(\binom{n}{n/k})$ time to try all possible assignments that are within Hamming distance n/k of a given assignment, the random walk algorithm takes only $O^*(k^{n/k})$ time instead, using the k-SAT instance to search over all those assignments. So over the whole space of 2^n assignments, you can think of the algorithm as "dividing out" a factor of $\binom{n}{n/k}$ and multiplying by $k^{n/k}$. This gives the $2^{n/O(k)}$ speedup.

Derandomization. The above uses a lot of randomness. We can actually derandomize the **Schon** algorithm by using a deterministic $k^{n/k}$ time algorithm for the local search (branch on all k choices of a literal to flip, for recursion depth of n/k), and using a subset $S \subseteq \{0,1\}^n$ of size $O^*(2^n/\binom{n}{n/k})$, with the property that every n-bit string is within n/k Hamming distance of some string in S (called a covering code); such subsets can be computed in time $O^*(2^n/\binom{n}{n/k})$. This was first reported in the literature by [DGHS00].

3.2 Schoening's actual algorithm (Optional material)

In the above algorithm **Schon**, we required ourselves to do a walk of length n/k, and each of the n/k variables we flipped must have been flipped correctly. What if we allowed ourselves to do a slightly longer random walk, where some of the flips are allowed to be incorrect? (We might walk away from a satisfying assignment a little bit, but then come back towards it.) Schoening's algorithm accounts for this possibility, giving a sharper running time than the above.

We present Schoening's algorithm for 3-SAT. It looks very similar to **Schon** given above, except that we will walk for $3n$ steps instead of $n/3$ steps.

Schon2(F):

```
Repeat $10(n+1)(4/3)^n$ times:
    Choose random assignment $A$.
    Repeat for $3n$ times:
        If $A$ satisfies $F$ return "SAT"
        If $A$ does not satisfy $F$, then
            Let $C$ be a falsified clause of $F$.
            Pick a random variable in $C$. Flip its value in $A$.
            [$(1/3)$ probability of choosing "correct" literal to flip]
    End repeat
End repeat
Return "UNSAT".
```

Theorem 3.4 *Suppose F is SAT. Then,* $\Pr[$**Schon2**(F) *returns "UNSAT"*$] < \exp(-10)$.

Let A^* be a satisfying assignment to F.

Claim 3.4 $Pr[$*Inner loop returns* A^**, starting from* $A] \geq (1/2)^{h(A,A^*)}/(n+1)$.

Recall that $h(x, y) =$ hamming distance between x and $y =$ number of bits in which x and y differ. Using the argument from **Schon**, we could just say that the above probability is at least $(1/3)^{h(A,A^*)}$: imagine that the algorithm chooses the correct literal each time. It does this with probability at least $1/3$, and it only has to do this for $h(A, A^*)$ times. However, this is too pessimistic: the probability can be much higher, because the algorithm could possibly make some mistakes in its variable flips, and end up correcting those flips later on. That's what Claim 3.4 takes advantage of.

Assume for now that Claim 3.4 holds.

Claim 3.5 $Pr_A[h(A, A^*) = k] = \binom{n}{k}/2^n$.

Note we already had this claim in our proof of our k-SAT algorithm. Now, assuming Claim 3.4,

$$Pr[\text{One repetition returns "SAT"}] \geq \frac{1}{2^n} \cdot \sum_{A \in \{0,1\}^n} Pr[\text{Inner loop returns } A^*, \text{ starting from } A].$$

Sorting the A's by Hamming distance from A^*, this equals

$$\sum_{k=0}^{n} Pr_A[h(A, A^*) = k] \cdot Pr[\text{Inner loop returns } A^*, \text{ starting from } A].$$

Applying Claim 3.4 and Claim 3.5, this quantity is at least

$$\sum_{k=0}^{n} \binom{n}{k} / 2^n \cdot (1/2)^k/(n+1).$$

By the binomial theorem, this equals

$$(3/4)^n/(n+1).$$

Exercise 3.3 *Verify that we applied the binomial theorem correctly.*

Therefore, after $10(n+1)(4/3)^n$ repetitions, there is probability less than $\exp(-10)$ of returning "UNSAT" when F is satisfiable.

Finally, we turn to the

Proof of Claim 3.4: Let $t = h(A, A^*)$. Consider the following event E:

> *Over a $3t$ step walk, we walk for t steps to the "right" on the line graph (make t "bad" choices of which variable to flip), and $2t$ steps to the "left" (make $2t$ "good" choices of which variable to flip).*

Note that $Pr[E] \geq (2/3)^t(1/3)^{2t} \cdot \binom{3t}{t}$.

Each time we step to the right, it costs probability $(2/3)$, when we step to the left it's prob $1/3$, and there are $\binom{3t}{t}$ possible ways to step t times to the right and $2t$ times to the left.

By introducing another 2^t factor, we can rewrite the above inequality as $Pr[E] \geq 1/2^t \cdot (1/3)^t(2/3)^{2t} \cdot \binom{3t}{t}$.

Let $A = (1/3)^t(2/3)^{2t}$ and $B = \binom{3t}{t}$. Now it suffices to show that $A \cdot B \geq 1/(3t+1)$. This is basically a combinatorial exercise.

Exercise 3.4 *Here's your cue! You have already seen the inequalities you will need in the proof (you can assume them).*

4 The Fastest Known 3-SAT Algorithms

4.1 Deterministic Algorithm

The fastest known deterministic algorithm for 3-SAT is a "full derandomization" of Schoening's algorithm. Moser and Scheder [MS11] replace all of the above random walk apparatus with deterministic choices! Their algorithm runs in $O((4/3 + \varepsilon)^n)$ time for all $\varepsilon > 0$.

4.2 Randomized Algorithm

The fastest known randomized upper bound for 3-SAT is currently due to Scheder [Sch24], running in about $O(1.307^n)$ time, slightly improving another recent bound of Hansen, Kaplan, Zamir, and Zwick [HKZZ19]. In particular, Scheder gave a new analysis of an older algorithm called PPSZ [PPSZ05] (named after its authors: Paturi, Pudlák, Saks, and Zane). You can think of PPSZ as a randomized branching algorithm: it randomly sets variables to random values, and aggressively checks if any variables are implied by its assignments. Here we sketch how PPSZ works.

Imagine a polynomial-time algorithm Simplify which given a k-SAT formula F, tries to simplify F as follows:

1. If there is a clause that contains both x and $\neg x$, remove it, as it is always satisfied.
2. If there is a 1-Clause (ℓ), set the literal ℓ to be true, remove any satisfied clauses, and remove all occurrences of $\neg\ell$ from the rest of the clauses. If any clause is empty, return "UNSAT".
3. Generalizing the above: try all subsets S of F with at most 100 3-clauses, and all satisfying assignments to the set S. If there is a variable y such that every satisfying assignment to S assigns y to a fixed value $v \in \{0, 1\}$, then set $y := v$, and simplify the formula as above.

Then, the PPSZ algorithm looks like the following.

```
PPSZ($F$):
Repeat until all variables are assigned:
    Until no more simplifications are possible, repeat: Simplify($F$)
    Pick a random unassigned variable $x$.
    Set $x$ to a random value.
End repeat.
```

The surprising theorem is:

Theorem 4.1 *Suppose F is satisfiable. Then* $\Pr[$*PPSZ returns a SAT assignment to* $F] \geq 1/(1.308)^n$.

Paturi, Pudlak, Saks and Zane (PPSZ) originally proved the above theorem for formulas F that have a unique satisfying assignment. Hertli [Her14] proved it for all F, and Scheder [Sch24] improved the probability. It was very recently shown by Scheder and Talebanfard that the PPSZ algorithm (in the above form) actually requires $2^{n-\Omega(n/k)})$ time on some instances [ST20]. That is, "Super Strong ETH" holds for the PPSZ algorithm.

4.3 Some Open Problems Regarding SAT

Besides the obvious open problems (improve the exponents) here are a couple of other interesting open issues.

- The best quantum algorithms for k-SAT run in $2^{n/2-n/O(k)}$ time, by directly applying Grover search to the PPSZ algorithm (which runs in polynomial time, and has success probability at least $1/2^{n-n/O(k)}$).

  Is there a faster quantum algorithm for k-SAT? Should we believe a Quantum Super-Strong ETH?

- Sometimes it is natural to have a domain for variables that is larger than just $\{0,1\}$. Define the (d,k)-CSP problem as follows. Each instance has n *variables*, where each variable can take a value from $[d] := \{1,\ldots,d\}$. Each instance also has m *constraints*, where each constraint is a function from $[d]^k$ to $\{True, False\}$, and it is defined on some subset of k of the variables. (For example, one can define 3-coloring on a domain of size d with constraints of size 2.)

  It turns out that all known (d,k)-CSP algorithms run in time at least $d^n/2^{O(n/k)}$, if not worse. This is a quite dismal improvement over the exhaustive search cost of d^n; note that all known k-SAT algorithms run in time at least $2^n/2^{O(n/k)}$.

  Is there an algorithm for (d,k)-CSP running in time $d^{n-n/f(k)}$, where f is an unbounded function? Is there some reason to believe that such an improvement is not possible? Intriguingly, if you look back at our previous algorithms where it was quite easy to get a $2^{n-n/f(k)}$-type bound for k-SAT, they don't get a $d^{n-n/f(k)}$-type bound for (d,k)-CSP.

  In related work, Koucký, Rödl, and Talebanfard [KRT21] show that, when each variable occurs at most r times for a constant r, there is an algorithm for (d,k)-CSP running in $d^{n-n/f(k,r)}$ time for some f. Their work also shows that such an algorithm also exists when the total number of constraints is at most rn for a constant r. So roughly speaking, this result says that if there is a sparsification lemma for (d,k)-CSP (which lets us reduce general (d,k)-CSPs to those in which there are at most $r_{k,\varepsilon}n$ constraints, in $d^{\varepsilon n}$ time) then we can resolve the above open problem.

5 A Natural Equivalent Version of SETH (Optional)

Finally, we consider variants on SETH. Recall that SETH states that for all $\varepsilon > 0$, there is a k such that k-SAT on n variables cannot be solved in $O(2^{(1-\varepsilon)n})$ time. We will show that SETH is equivalent to the claim that satisfiability on CNFs with $O(n)$ clauses (of *any width*) cannot be solved in substantially less than 2^n time.

Consider the following More-Believable SETH (MBSETH):

> For all $\varepsilon > 0$, there is a c such that CNF-SAT on n variables and cn clauses cannot be solved in $O(2^{(1-\varepsilon)n})$ time.

Note, there is **no** restriction on the width of clauses in MBSETH. MBSETH looks more believable, as the Sparsification Lemma shows that SETH implies MBSETH (we can always reduce k-SAT to the case of $O(n)$ clauses). We show that SETH and MBSETH are actually equivalent, a result proved by Calabro, Impagliazzo and Paturi [CIP06]. To show this, we will prove that MBSETH implies SETH:

Theorem 5.1 *Suppose there is a $\delta < 1$ such that for all constant k, k-SAT is in $O(2^{\delta n})$ time (SETH is false). Then there is a $\gamma < 1$ such that for every $c \geq 1$, CNF-SAT with cn clauses is in $O(2^{\gamma n})$ time.*

The reduction itself is an interesting branching algorithm. We start with a CNF that has cn clauses of any width, and we want to reduce it to instances of k-SAT. How might we go about this? Take a clause of length greater than k in our starting CNF. We want to get rid of it somehow. We can do this by branching. First, we

1. assert that at least one of the first k literals is true (replacing the long clause with a k-clause), and in the other branch we

2. assert that the first k literals of the clause are all false.

In the first branch, we get rid of a long clause, and we can assert a k-clause instead. In the second branch we get to set k variables in the formula: a huge reduction. Here are the details.

Proof. Suppose there is a $\delta < 1$ such that for all constant k, k-SAT is in $O(2^{\delta n})$ time.

Let F be a CNF formula with n variables and cn clauses, with $c \geq 1$. We will show how to use an $O(2^{\delta n})$-time k-SAT algorithm to also solve SAT for F.

Let k be a **parameter** to set later, as a function of c.

Our algorithm for satisfiability runs as follows:

If all clauses have length at most k, then solve the instance using the $O(2^{\delta n})$-time k-SAT algorithm.
Else, take a clause more than k literals, e.g., $C = (x_1 \vee \ldots \vee x_k \vee \ldots)$.
Recursively call the algorithm on two instances:
(1) F with C replaced by $(x_1 \vee \ldots \vee x_k)$, and (2) F with $x_1 := 0, \ldots, x_k := 0$.
If both are UNSAT then return UNSAT, else return SAT.

There are at most n/k branches of type (2) where k variables are set, and at most cn branches of type (1) where a "long" clause is shortened to be of length k. Once we reach a k-SAT instance, we run our $O(2^{\delta n'})$ time k-SAT algorithm on the remaining n' variables. Suppose in some sequence of recursive calls (from the start until we call the k-SAT algorithm) we call the algorithm i times on branches of type (2). Then the running time when we call the k-SAT algorithm is at most $O(2^{\delta(n-ik)})$, because for every branch of type (2) we remove k variables from the formula.

For simplicity, let's only look at the leaves that take exactly cn branches of type (1) (in general, we'd also want to sum over all choices $j = 0, \ldots, cn$ where we take j branches of type (1), but considering these only introduces a poly(n) factor into the running time).

The total running time then is at most

$$\sum_{i=0}^{n/k} \binom{cn+i}{i} \cdot 2^{\delta(n-ik)},$$

where we are summing over all possible $i = 0, \ldots, nk/k$ such that we make i branches of type (2) before the k-SAT algorithm is called: there are $\binom{cn+i}{i}$ possible "leaves" of the branching tree where there are i branches of type (2) on the path from the root to the leaf, and for each of those leaves, there are at most $n - ik$ variables, so the k-SAT algorithm takes $2^{\delta(n-ik)}$ time.

Factoring out $2^{\delta n}$, we have

$$\sum_{i=0}^{n/k} \binom{cn+i}{i} \cdot 2^{\delta(n-ik)} \leq 2^{\delta n} \cdot \sum_{i=0}^{n/k} \binom{cn+i}{i} 2^{-\delta ik}.$$

In our case, $c \geq 1$, so $\binom{cn+i}{i}$ is maximized for the largest setting of i, $i = n/k$, and so the runtime is at most

$$2^{\delta n} \binom{cn+n/k}{n/k} \cdot \sum_{i=0}^{n/k} 2^{-\delta ik} \leq O\left(2^{\delta n} \cdot \binom{cn+n/k}{n/k}\right),$$

as the infinite geometric series converges. Recalling $\binom{N}{K} \leq (eN/K)^K$, we find that the runtime is big-O of

$$2^{\delta n} e^{n/k} \cdot \left(\frac{cn + n/k}{n/k}\right)^{n/k} = 2^{\delta n} e^{n/k} (ck+1)^{n/k} = 2^{\delta n} e^{n/k} 2^{n \cdot \log(ck+1)/k}.$$

For simplicity, let's ignore the $e^{n/k}$ term, as it's negligible compared to $2^{n \cdot \log(ck+1)/k}$.

We want to set k so that $\log(kc+1)/k < \varepsilon$ for an $\varepsilon > 0$ such that $\delta + \varepsilon < 1$, so that we can set $\gamma := \delta + \varepsilon < 1$. Then it would suffice if $\log(2kc) < k\varepsilon$, and so $\log(k) + \log(2c) < k\varepsilon$ and $k\varepsilon - \log(k) > \log(2c)$.

If we set k large enough so that $k\varepsilon/2 \geq \log k$, then we'd just need $k\varepsilon > 2\log(2c)$.

So k needs to be $\geq \max\{2\log k/\varepsilon, 2\log(2c)/\varepsilon\}$. If we set $k = Z \cdot 2\log(2c)/\varepsilon$ for $Z \geq 1$, then $Z\log(2c) = \varepsilon k/2$, and $\log(k) = \log(2Z/\varepsilon) + \log\log(2c)$. Since we want $k\varepsilon/2 \geq \log k$, we thus want $Z\log(2c) \geq \log(2Z/\varepsilon) + \log\log(2c)$ and $Z\log(c) \geq \log(Z) + \log(1/\varepsilon) + \log\log(2c)$. For all $Z \geq 4$, $Z/2 \geq \log(Z)$, so if we set $Z \geq 4$, we would only need $Z\log(c) \geq 2(\log(1/\varepsilon) + \log\log(2c))$. Similarly, whenever c is a large enough constant, $\log(c) > 4\log\log(2c)$, so we only really need $Z\log(c) \geq 4\log(1/\varepsilon)$, so we can set $Z = 4\log(1/\varepsilon)$ and the inequality will be true.

Hence if we set k proportional to $\log(1/\varepsilon)\log(c)/\varepsilon$, using our conjectured $O(2^{\delta n})$ time algorithm for k-SAT, we obtain an $O(2^{(\delta+\varepsilon)n})$ time algorithm for CNF-SAT with cn clauses, for any small $\varepsilon > 0$.

How small does $\varepsilon > 0$ need to be? Here's one way to calculate an ε. Note that if $\delta < 1$, then $(\delta + 1)/2$ is also less than 1. Thus if we set ε such that $\delta + \varepsilon = (\delta + 1)/2$, i.e., $\varepsilon = (1 - \delta)/2$, we get $\gamma = \delta + \varepsilon < 1$. □

References

[BE05] Richard Beigel and David Eppstein. 3-coloring in time o(1.3289^n). *J. Algorithms*, 54(2):168–204, 2005.

[CIP06] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. A duality between clause width and clause density for SAT. In *21st Annual IEEE Conference on Computational Complexity (CCC 2006), 16-20 July 2006, Prague, Czech Republic*, pages 252–260. IEEE Computer Society, 2006.

[DGHS00] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, and Uwe Schöning. Deterministic algorithms for *k*-sat based on covering codes and local search. In *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 236–247. Springer, 2000.

[Her14] Timon Hertli. 3-SAT faster and simpler - Unique-SAT bounds for PPSZ hold in general. *SIAM J. Comput.*, 43(2):718–729, 2014.

[HKZZ19] Thomas Dueholm Hansen, Haim Kaplan, Or Zamir, and Uri Zwick. Faster *k*-sat algorithms using biased-ppsz. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 578–589. ACM, 2019.

[KRT21] Michal Koucký, Vojtech Rödl, and Navid Talebanfard. A separator theorem for hypergraphs and a CSP-SAT algorithm. *Log. Methods Comput. Sci.*, 17(4), 2021.

[MS85] Burkhard Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discret. Appl. Math.*, 10(3):287–295, 1985.

[MS11] Robin A. Moser and Dominik Scheder. A full derandomization of Schöning's k-SAT algorithm. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 245–252. ACM, 2011.

[Pap91] Christos H. Papadimitriou. On selecting a satisfying truth assignment (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 163–169. IEEE Computer Society, 1991.

[PPSZ05] Ramamohan Paturi, Pavel Pudlák, Michael E. Saks, and Francis Zane. An improved exponential-time algorithm for k-sat. *J. ACM*, 52(3):337–364, 2005.

[Sch99] Uwe Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 410–414. IEEE Computer Society, 1999.

[Sch02] Uwe Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002.

[Sch24] Dominik Scheder. PPSZ is better than you think. *TheoretiCS*, 3, 2024.

[ST20] Dominik Scheder and Navid Talebanfard. Super strong ETH is true for PPSZ with small resolution width. In Shubhangi Saraf, editor, *35th Computational Complexity Conference, CCC 2020, July 28-31, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 169 of *LIPIcs*, pages 3:1–3:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[VW19] Nikhil Vyas and R. Ryan Williams. On Super Strong ETH. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 406–423. Springer, 2019.

[Wil19] R. Ryan Williams. Some estimated likelihoods for computational complexity. In Bernhard Steffen and Gerhard J. Woeginger, editors, *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*, pages 9–26. Springer, 2019.